Universidad ORT Uruguay
Facultad de Ingeniería

Diseño de Aplicaciones 2

Obligatorio 2

Martin Civetta (271063)
Ignacio Etcheverry (279855)

https://github.com/NACHO9999/OB-DAP2

10 de Junio del 2024

Descripción Técnica del Sistema de Gestión de Edificios

Este sistema está diseñado como una herramienta integral para la gestión de edificios, orientado tanto al mercado local como internacional. Su arquitectura avanzada y las tecnologías implementadas facilitan la administración eficaz de propiedades por parte de administradores y encargados de empresas constructoras.

Arquitectura y Tecnologías

- La aplicación se basa en una arquitectura de microservicios, utilizando Angular para el front-end y ASP.NET Core 8.0 para el back-end. La gestión de datos se lleva a cabo mediante Microsoft SQL Server Express 2019, apoyado en Entity Framework Core 8.0 para el mapeo de datos.

Funcionalidades Clave

1. Interfaz de Usuario Completa:
    - Desarrollada en Angular, esta interfaz de página única ofrece una experiencia de usuario fluida que integra todas las funcionalidades preexistentes y las nuevas, mejorando la interacción del usuario con el sistema.

2. Rol de Admin de Empresa Constructora:

    - Gestión de Empresas Constructoras: Los usuarios con este rol pueden crear sus propias empresas en el sistema, manejar edificios y asignar encargados.
    - Importación de Edificios: Se ha añadido la capacidad de importar datos de edificios desde sistemas externos utilizando adaptadores desarrollados por terceros, lo que permite la integración de múltiples formatos como XML, JSON y CSV.

3. Extensibilidad y Modularidad:
    - La estructura del sistema permite incorporar nuevas funcionalidades sin recompilar el código fuente completo, facilitando la adaptación y expansión continua.

Consideraciones de Diseño

- Mantenibilidad y Escalabilidad: El diseño modular del sistema favorece su mantenimiento y actualización, permitiendo cambios en componentes específicos sin afectar al conjunto.
- Seguridad y Roles de Usuario: Se establecen roles claros dentro del sistema para asegurar que las operaciones críticas solo sean ejecutadas por usuarios autorizados.
- Usabilidad: Siguiendo principios de diseño de interfaz reconocidos, se ha creado una interfaz intuitiva y fácil de manejar para mejorar la eficiencia del usuario y reducir errores.

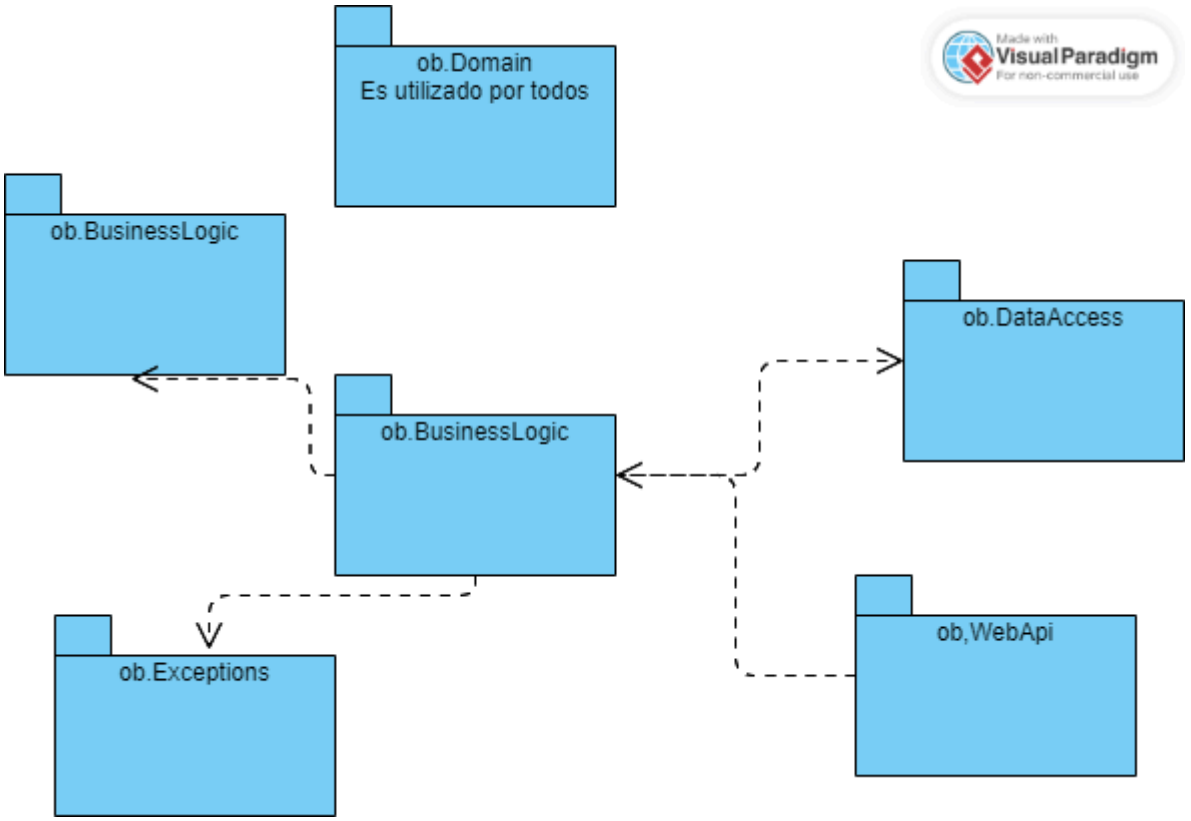
¿Qué hace la solución?

Nuestra solución es una aplicación para la gestión de edificios, permitiendo a los administradores y encargados manejar propiedades y coordinar tareas de mantenimiento. Las principales características incluyen:

1. **Gestión de Usuarios y Roles:** Creación y gestión de empresas constructoras, edificios y usuarios de mantenimiento.
2. **Gestión de Edificios:** Alta, modificación y eliminación de edificios, y su importación desde sistemas externos.
3. **Gestión de solicitudes:** Creación, seguimiento y reporte de solicitudes de mantenimiento.
4. **Interfaz de Usuario:** Desarrollada en Angular, ofreciendo una experiencia fluida e intuitiva.

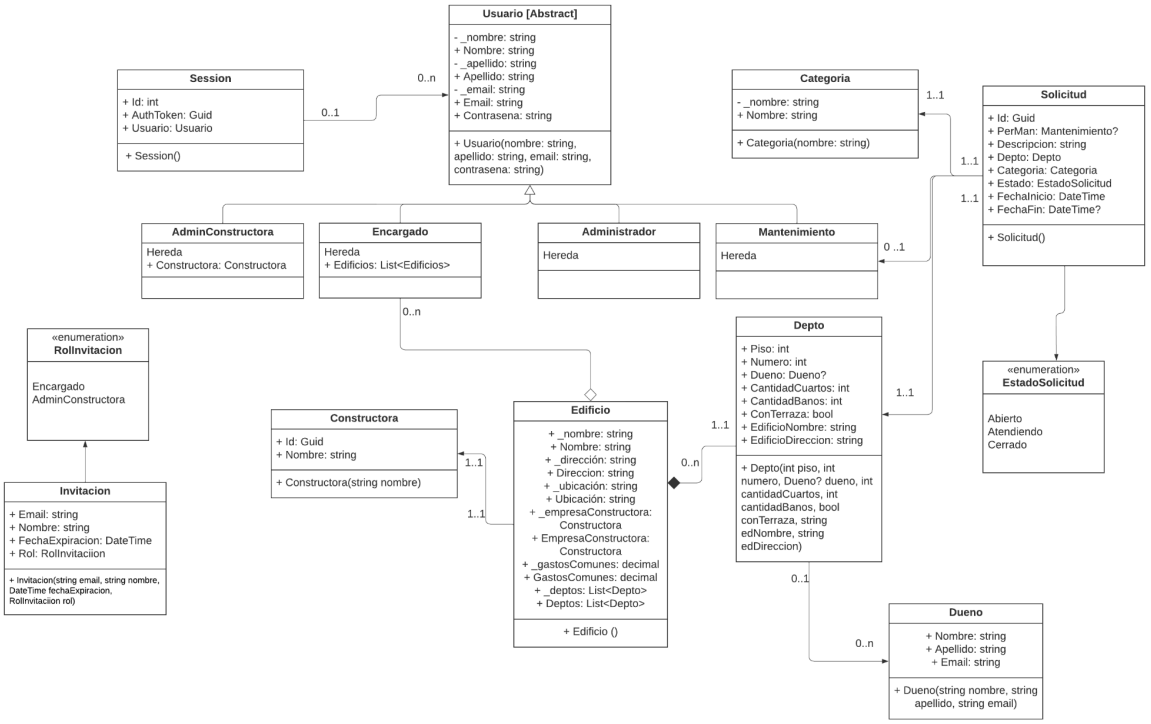
Errores Conocidos y Funcionalidades No Implementadas

1. **Errores Conocidos:**
    - **Lentitud en Listados Grandes:** Problemas de performance al listar grandes cantidades de datos.
    - **Validación de Importaciones:** Problemas en la validación de algunos formatos de importación de edificios.
    - **Notificaciones de Estado:** Inconsistencias en las notificaciones en tiempo real sobre el estado de las solicitudes.
2. **Funcionalidades No Implementadas:**
    - **Exportación de Datos:** Falta de funcionalidad para exportar datos a CSV o PDF.
    - **Integración Completa de Adaptadores de Terceros:** No todos los tipos de fuentes están completamente integrados.

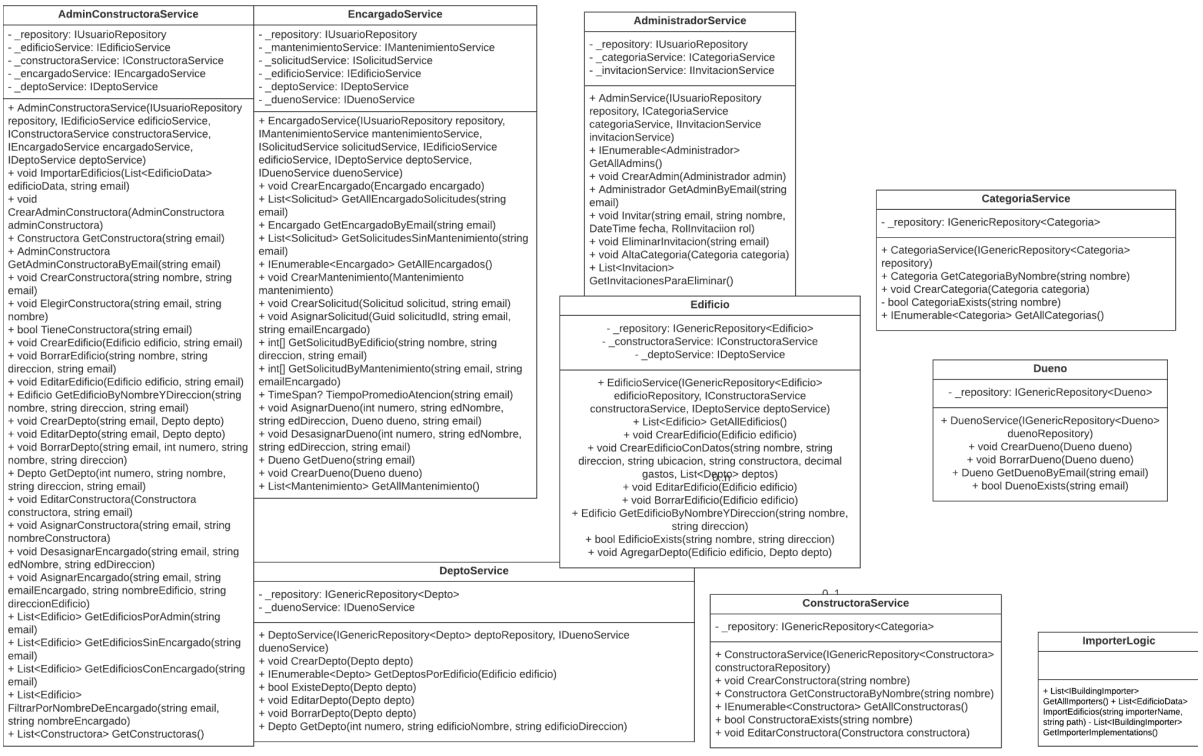


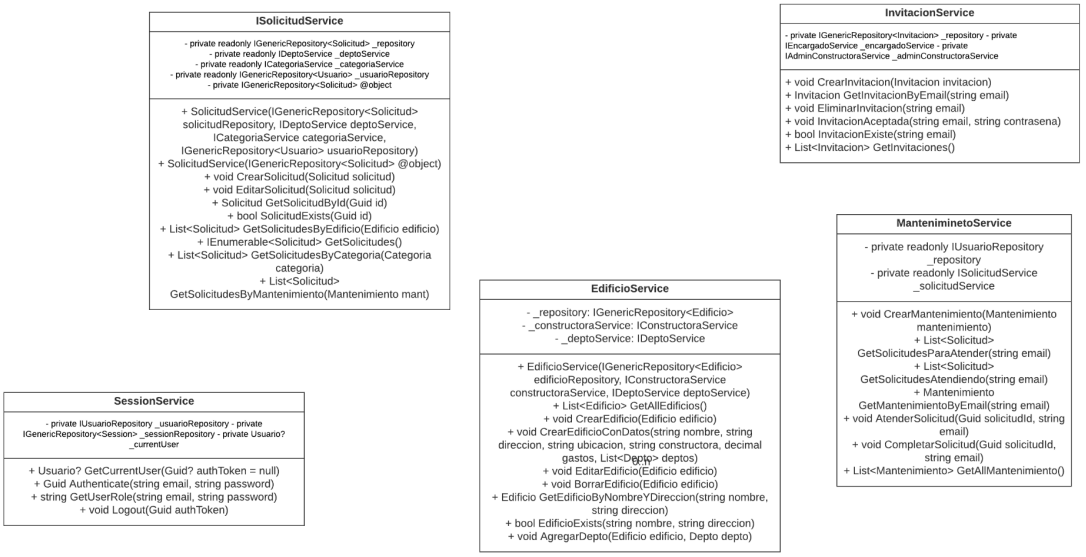
- En realidad todos los paquetes no dependen directamente de los otros paquetes. Dependen de un paquete de interfaces que se titulan igual al nombre del paquete que están apuntando

(en excepción de exceptions). Los paquetes de interfaces se llaman igual solo que con una I delante. No los mostramos en este diagrama para facilitar la legibilidad

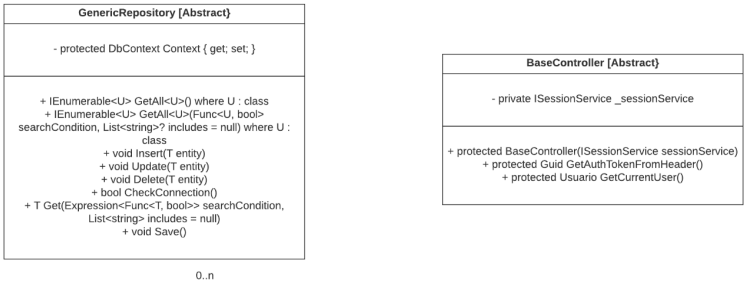


- En este caso, son las clases de dominó y no se puede explicar mucho más de lo que se ve. Hicimos una clase abstracta de usuario para facilitar las operaciones e validación de usuario





- En este caso usamos una abstracción del repositorio genérico para cada uno de los servicios para tratar con sus propias operaciones. A su vez, le pasamos abstracciones de los otros servicio para aquellas clases que tengan funcionalidades sobre otras



- Para data access hicimos este repositorio genérico. No incluimos a las otras clases porque todas heredan de esta sin modificación salvo de UsuarioRepositorio, que tiene un método EmailExiste que facilita la lógica de creación de usuarios.

- En webapi hicimos los controllers que llaman las funcionalidades declaradas en businesslogic. Salvo por esta clase abstracta que la usamos para facilitar la validación de acción propia de usuarios

En nuestra solución, implementamos jerarquías de herencia para mejorar la extensibilidad y reutilización del código. A continuación, se detallan las principales jerarquías de herencia utilizadas:

Jerarquía de la Clase Usuario

- **Clase Base: Usuario**
    - **Descripción:** Esta es la clase base de todos los tipos de usuarios en el sistema. Contiene propiedades comunes como Nombre, Apellido, Email, y Contraseña.
    - **Subclases:**
        - **Administrador**
            - **Descripción:** Hereda de Usuario y representa a los usuarios administradores generales del sistema.
        - **AdminConstructora**
            - **Descripción:** Hereda de Usuario y añade propiedades y métodos específicos para la administración de empresas constructoras.
            - **Relación:** Cada AdminConstructora puede estar asociado a una Constructora.
        - **Encargado**
            - **Descripción:** Hereda de Usuario y representa a los encargados de edificios.
            - **Relación:** Cada Encargado puede estar asociado a múltiples Edificios.
        - **Mantenimiento**
            - **Descripción:** Hereda de Usuario y representa al personal de mantenimiento.
            - **Relación:** Cada Mantenimiento puede estar asignado a múltiples Solicitudes de mantenimiento.

Jerarquía de la Clase Solicitud

- **Clase Base: Solicitud**
    - **Descripción:** Esta clase representa una solicitud de servicio en el sistema y contiene propiedades como Descripcion, Estado, Categoria, y Depto.
    - **Subclases:**
        - **SolicitudMantenimiento**
            - **Descripción:** Hereda de Solicitud y añade propiedades específicas para solicitudes de mantenimiento, como CostoTotal y FechaInicio.

Jerarquía de la Clase Edificio

- **Clase Base: Edificio**
    - **Descripción:** Representa un edificio en el sistema, con propiedades como Nombre, Direccion, GastosComunes.
    - **Subclases:**

    - **EdificioResidencial**
        - **Descripción:** Hereda de Edificio y añade propiedades específicas para edificios residenciales, como NumeroDeDepartamentos.
    - **EdificioComercial**
        - **Descripción:** Hereda de Edificio y añade propiedades específicas para edificios comerciales, como NumeroDeOficinas.

Justificación del Uso de Herencia

El uso de herencia en nuestro diseño facilita la extensibilidad y el mantenimiento del sistema. Nos permite definir comportamientos comunes en clases base y extender funcionalidades específicas en subclases. Esto reduce la duplicación de código y mejora la organización y claridad del mismo, haciendo que sea más fácil de entender y modificar.

Modelo de Tablas de la Base de Datos

Configuración General

- El proyecto utiliza Entity Framework Core con el enfoque Code First para definir y manejar la base de datos. Este enfoque permite especificar el modelo de la base de datos a través de clases en C#, lo que facilita la abstracción y el mantenimiento del código.

Clases y Tablas

- **Usuarios**: La tabla Usuarios actúa como la tabla base para diferentes tipos de usuarios en el sistema, utilizando herencia para diferenciar entre Administrador, AdminConstructora, Encargado, y Mantenimiento. Cada uno de estos tipos de usuario hereda de la clase Usuario, lo que permite reutilizar campos comunes y facilitar el manejo de diferentes roles.
- **Edificios y Constructoras**: Cada Edificio está relacionado con una Constructora, representando la relación entre los edificios y las empresas constructoras que los administran.
- **Solicitudes, Departamentos, y Mantenimiento**: Las Solicitudes están relacionadas con Departamentos y pueden estar asignadas a personal de Mantenimiento. Esto modela las solicitudes de servicios y su gestión dentro de los departamentos específicos.

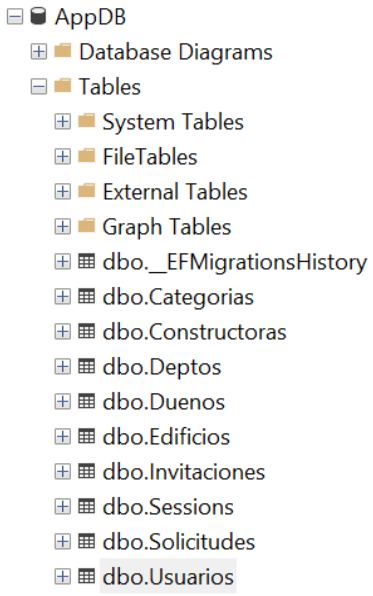
Relaciones y Llaves

- **Llaves Primarias y Compuestas**: Algunas tablas utilizan llaves primarias compuestas, como Depto, que utiliza Numero, EdificioNombre, y EdificioDireccion para identificar de forma única cada departamento.
- **Relaciones One-to-Many y Many-to-One**: Utilizamos configuraciones como .HasOne() y .WithMany() para definir relaciones entre las tablas, por ejemplo, un Edificio que puede tener múltiples Departamentos, pero cada Departamento está asociado a un único Edificio.

Configuración de Propiedades Específicas

- **Tipos de Columnas y Requerimientos**: Utilizamos métodos como .HasColumnType() para definir tipos específicos de SQL para campos como GastosComunes en la tabla Edificio, asegurando que el campo se maneje adecuadamente en la base de datos.

Inyección de Dependencias y Configuración del Contexto



- **DbContextOptions**: La clase AppContext se configura mediante DbContextOptions, lo que permite la inyección de dependencias y la configuración flexible del contexto durante la ejecución.

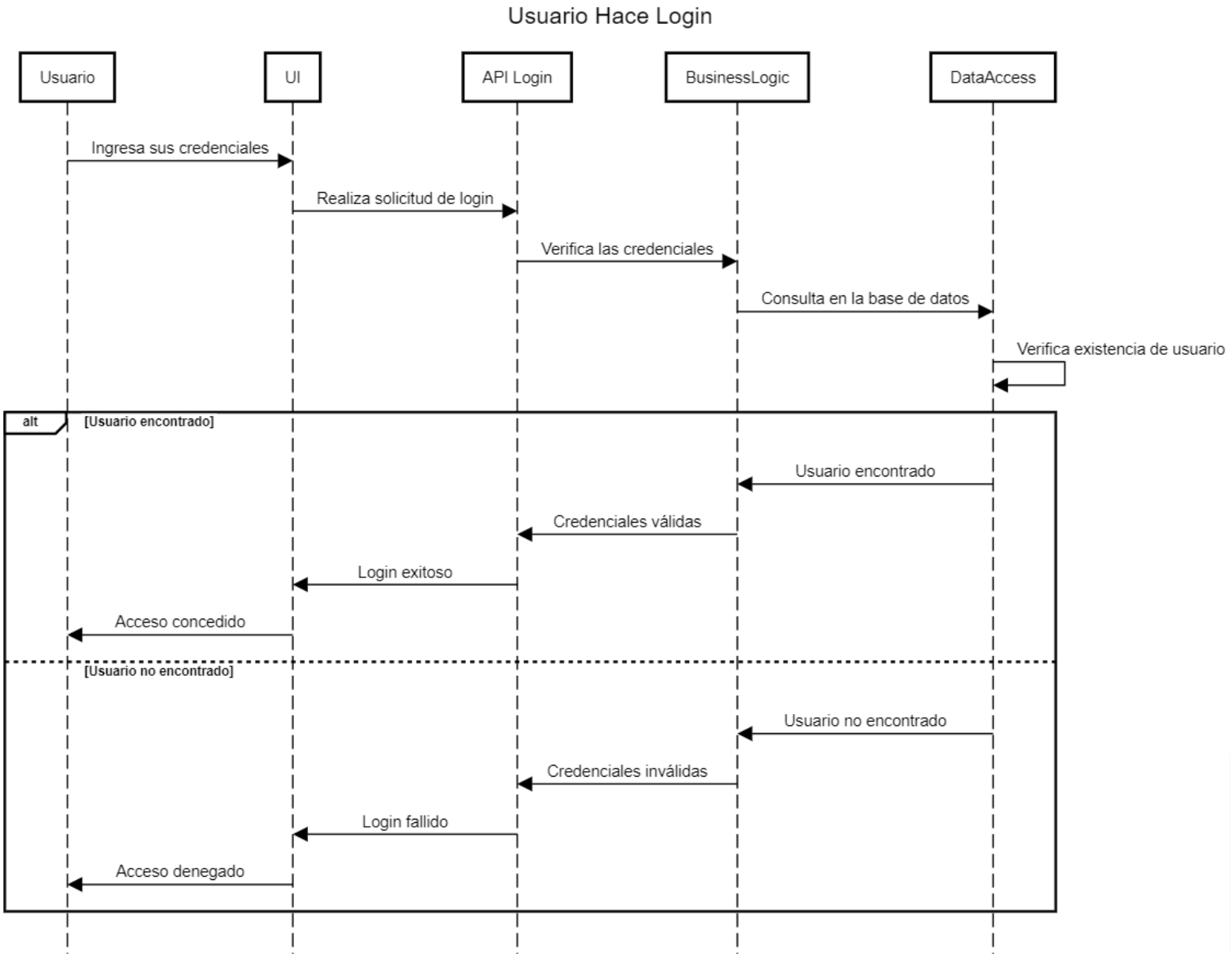
Consideraciones para el Desarrollo y Mantenimiento

Utilizando el enfoque Code First, facilitamos la integración entre el desarrollo de la base de datos y el código de la aplicación, lo que permite cambios más ágiles y una mejor alineación con los requisitos del negocio. Además, el uso de herencia y relaciones complejas en el modelo de datos ayuda a mantener la integridad y coherencia del diseño.

Diagramas de Interacción

Funcionalidad de Inicio de Sesión

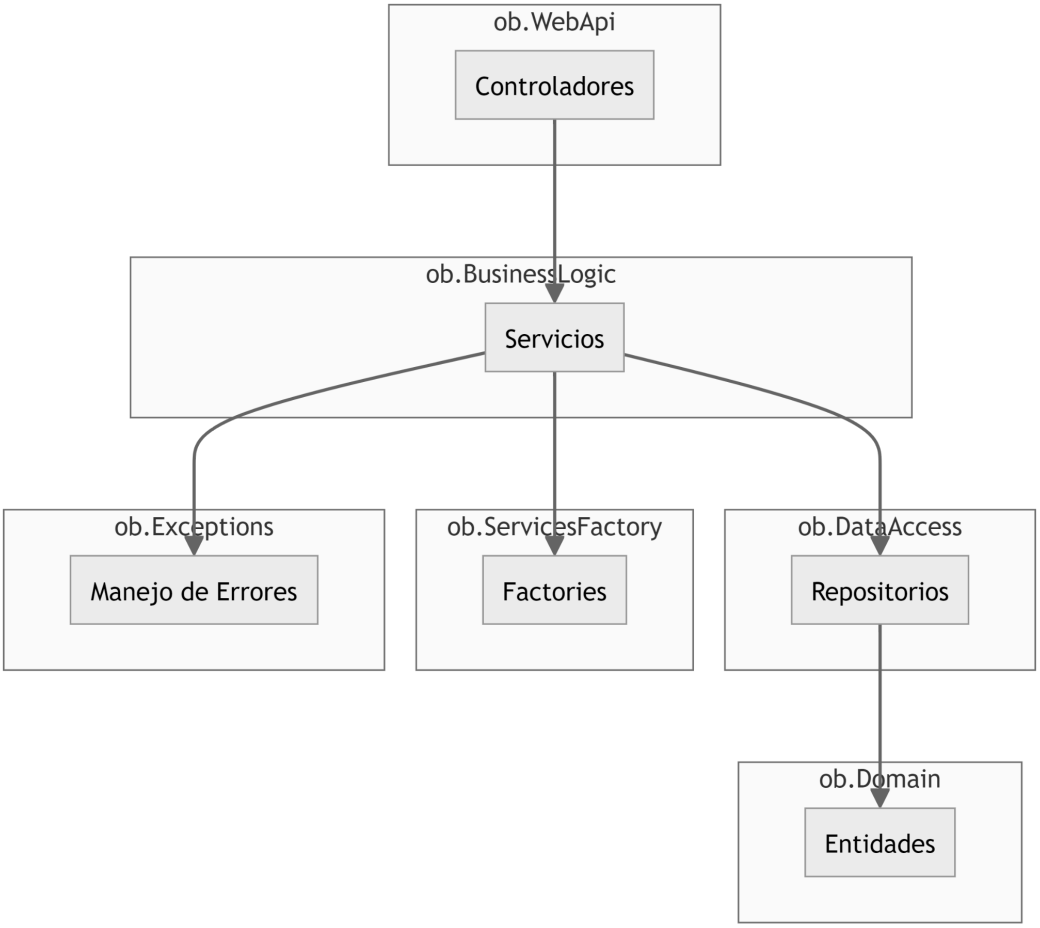
El siguiente diagrama de interacción muestra el proceso de inicio de sesión de un usuario en el sistema. Este diagrama ilustra las clases involucradas y las interacciones entre ellas para autenticar al usuario.



Descripción del Proceso:

1. Usuario ingresa sus credenciales a través de la UI (Interfaz de Usuario).
2. La UI realiza una solicitud de login al API Login.
3. El API Login envía las credenciales al BusinessLogic para su verificación.
4. El BusinessLogic consulta la base de datos a través del DataAccess para verificar la existencia del usuario con las credenciales proporcionadas.
5. DataAccess comprueba si el usuario existe en la base de datos.
    - Si el usuario es encontrado, DataAccess devuelve la confirmación al BusinessLogic.
    - Si el usuario no es encontrado, DataAccess informa al BusinessLogic que las credenciales son inválidas.
6. Dependiendo del resultado de la verificación:
    - Si las credenciales son válidas, BusinessLogic informa al API Login y este a la UI que el login fue exitoso, otorgando acceso al usuario.
    - Si las credenciales son inválidas, BusinessLogic informa al API Login y este a la UI que el login falló, denegando el acceso al usuario.

Este diagrama y proceso aseguran que las credenciales del usuario sean verificadas de manera segura y eficiente, integrando las diferentes capas de la arquitectura del sistema.



Justificación de la División en Componentes

1. **Modularidad:**
    - Dividimos la solución en componentes independientes para asegurar que cada parte del sistema sea responsable de una única funcionalidad. Esto facilita el mantenimiento y la actualización de componentes individuales sin afectar a otros.
2. **Separation of Concerns (Separación de Responsabilidades):**
    - **ob.WebApi:**
        - Este componente maneja las interacciones con los clientes a través de controladores y endpoints API. Al separar esta lógica, aseguramos que los cambios en la lógica de presentación no afecten la lógica de negocios ni el acceso a datos.
    - **ob.BusinessLogic:**
        - Contiene la lógica de negocio del sistema. Este componente es responsable de procesar las solicitudes del usuario, aplicar reglas de negocio y coordinar las interacciones entre los diferentes repositorios de datos.
    - **ob.DataAccess:**
        - Maneja todas las operaciones de acceso a datos. Al encapsular el acceso a datos en este componente, podemos cambiar fácilmente la base de datos subyacente o la tecnología de acceso a datos sin afectar otras partes del sistema.

- **ob.Domain:**
  - Define las entidades del dominio del sistema. Al centralizar la definición de entidades, mantenemos un modelo coherente y reutilizable que puede ser compartido por otros componentes.
- **ob.ServicesFactory:**
  - Proporciona fábricas para crear instancias de servicios y otros objetos. Este componente ayuda a gestionar la creación de objetos complejos y garantiza que las dependencias se inyecten correctamente.
- **ob.Exceptions:**
  - Centraliza el manejo de excepciones y errores. Al tener un componente dedicado para las excepciones, podemos manejar los errores de manera uniforme y centralizada, mejorando la robustez del sistema.

3. **Escalabilidad:**
   - La modularidad permite escalar cada componente de manera independiente. Por ejemplo, podemos desplegar múltiples instancias de ob.WebApi para manejar más solicitudes de usuario sin necesidad de escalar otros componentes que no son el cuello de botella.
4. **Extensibilidad:**
   - Cada componente puede ser extendido o reemplazado con una mínima interrupción del sistema. Por ejemplo, podemos añadir nuevos servicios en ob.BusinessLogic o nuevos repositorios en ob.DataAccess sin afectar la estructura general del sistema.
5. **Reusabilidad:**
   - Al dividir la solución en componentes, facilitamos la reutilización de código en diferentes contextos. Por ejemplo, las entidades en ob.Domain pueden ser reutilizadas en otros proyectos o servicios sin modificación.

Esta estructura modular no solo mejora la mantenibilidad y escalabilidad del sistema, sino que también facilita la colaboración entre diferentes equipos de desarrollo, permitiendo que cada equipo trabaje en componentes específicos sin interferir con otros.

Principios de Diseño Aplicados

- Principio de Responsabilidad Única (SRP): Cada clase en el sistema tiene una sola responsabilidad. Por ejemplo, los servicios en *categoriaService.cs* y *constructoraService.cs* gestionan las operaciones relacionadas únicamente con su contexto, ya sea administración general o específica de constructoras. Inevitablemente, hay clases que al interactuar con elementos de otras clases, heredan sus funcionalidades. En estos casos, pasamos los servicios de las clases a su servicio como atributo para utilizar esas operaciones, como las operaciones de edificios en la clase de admin constructora. Esto se hace para facilitar el entendimiento del código en las operaciones del usuario.

- Principio Abierto/Cerrado (OCP): Las clases están abiertas para extensión pero cerradas para modificación. Esto se observa en el uso de herencia y polimorfismo en los componentes y servicios, permitiendo añadir nuevas funcionalidades sin alterar las clases existentes.

- Inversión de Dependencias (DIP): Se depende de abstracciones, no de concreciones. Esto se implementa a través de la inyección de dependencias en Angular, facilitando el manejo de la *businessLogic* y la interacción con *dataAccess* sin acoplar fuertemente los componentes a implementaciones específicas. En todo caso en que una clase dependa de clases de otro paquete, e incluso cuando dependa de clases del mismo paquete, proporcionamos interfaces para mejorar la abstracción y reducir la dependencia entre clases.

Patrones de Diseño Utilizados

- Model-View-Controller (MVC): Claramente implementado donde Angular maneja las vistas, ASP.NET Core actúa como el controlador, y Entity Framework maneja el modelo. Esta separación ayuda a organizar el código de manera eficiente y mejora la testabilidad.

- Strategy: Este patrón se puede ver en la implementación de diferentes estrategias de autenticación y autorización, donde diferentes roles y permisos determinan la estrategia de acceso a los recursos.

Métricas de Calidad del Código

- Complejidad Ciclomática: Se mantiene baja en la mayoría de los métodos para evitar estructuras de control complicadas. Esto se logra limitando el número de rutas de ejecución y utilizando declaraciones condicionales simples.

- Cohesión: Los módulos tienen alta cohesión, realizando tareas bien definidas y relacionadas. Esto facilita el mantenimiento y la comprensión del código.

- Acoplamiento: Bajo entre módulos, lo que significa que los cambios en una parte del sistema tienen un impacto mínimo en otras partes. Esto se logra a través de la inyección de dependencias y el uso de interfaces para definir contratos claros entre diferentes partes del sistema.

Consideraciones de Diseño Adicionales

- Documentación y Nomenclatura: El código utiliza nombres descriptivos para clases, métodos y variables, lo que facilita la comprensión rápida del propósito de cada componente sin necesidad de comentarios extensos.

- Pruebas Unitarias y TDD (Desarrollo Guiado por Pruebas): Se observa en los archivos backend, donde cada funcionalidad principal tiene pruebas asociadas, asegurando que el código cumpla con los requisitos antes de su implementación definitiva y facilitando futuras modificaciones sin temor a romper funcionalidades existentes. Es relevante también mencionar que los cambios entre el primer y segundo obligatorio presentaron nuevos retos para mantener las pruebas TDD al 100%. Aunque estos ajustes eran necesarios para mejorar y expandir nuestro sistema, provocaron una reducción temporal en la cobertura de las pruebas.

Este enfoque no solo prepara el sistema para futuras expansiones y facilita la gestión de nuevas características o cambios, sino que también asegura que cualquier desarrollador pueda entender rápidamente las operaciones y estructura del código para continuar eficazmente con su desarrollo.

Mecanismos de Extensibilidad

Para la extensibilidad solicitada en la funcionalidad del nuevo obligatorio, el sistema de gestión de edificios fue diseñado con varios mecanismos que facilitan la incorporación de nuevas características y la adaptación a diferentes entornos o requisitos. Aca se describen los mecanismos clave utilizados:

Inyección de Dependencias (DI):

- Descripción: El sistema utiliza fuertemente la inyección de dependencias, especialmente en el backend con ASP.NET Core y en el frontend con Angular. Esta técnica desacopla la creación de objetos de su uso, permitiendo que las implementaciones sean intercambiables sin modificar el código que las utiliza.
- Beneficio: Facilita la adición o modificación de componentes (como servicios o controladores) sin necesidad de alterar las clases que dependen de estos. Por ejemplo, si se necesita cambiar la lógica de acceso a datos o introducir una nueva estrategia de autenticación, se puede hacer sin afectar al resto del sistema.

Patrones de Diseño Modular:

- Descripción: El sistema está construido siguiendo un modelo modular, especialmente en el frontend donde Angular permite la creación de módulos que pueden ser cargados de manera dinámica. Esto se complementa con la estructura de servicios y controladores en el backend que son fácilmente extensibles.
- Beneficio: Permite a los desarrolladores añadir nuevas funcionalidades como módulos independientes que se integran con el sistema principal sin complicaciones, manteniendo una organización clara y separada del código.

APIs y Contratos Bien Definidos:

- Descripción: El backend proporciona una API REST bien definida que actúa como una interfaz clara entre el frontend y el backend. Cada endpoint de la API está diseñado para ser lo más genérico posible, facilitando la extensión y modificación.
- Beneficio: Nuevos clientes o nuevos componentes del frontend pueden ser desarrollados y conectados al backend sin necesidad de cambios en el servidor, siempre y cuando respeten los contratos establecidos.

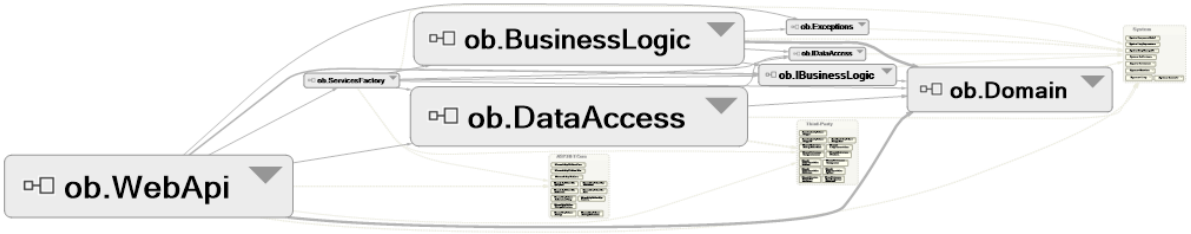
Documentación Clara y Accesible:

- Descripción: Todo el sistema está acompañado de documentación detallada, incluyendo la especificación de la API, los modelos de datos y la arquitectura del sistema.
- Beneficio: Proporciona una guía esencial para agregar nuevas características o al intentar entender el sistema para modificaciones futuras. La documentación actúa como un contrato que facilita la extensibilidad y mantenimiento.

Estos mecanismos no solo aseguran que el sistema sea adaptable y capaz de evolucionar en respuesta a nuevos requisitos o tecnologías, sino que también protegen las inversiones en desarrollo al reducir el riesgo de obsolescencia y minimizar el esfuerzo requerido para futuras expansiones.

Análisis del Diagrama de Dependencias

El diagrama de dependencias de nuestro sistema muestra una arquitectura bien organizada y modular, sin ciclos de dependencia, lo cual es fundamental para mantener la mantenibilidad y extensibilidad del software.



Descripción General:

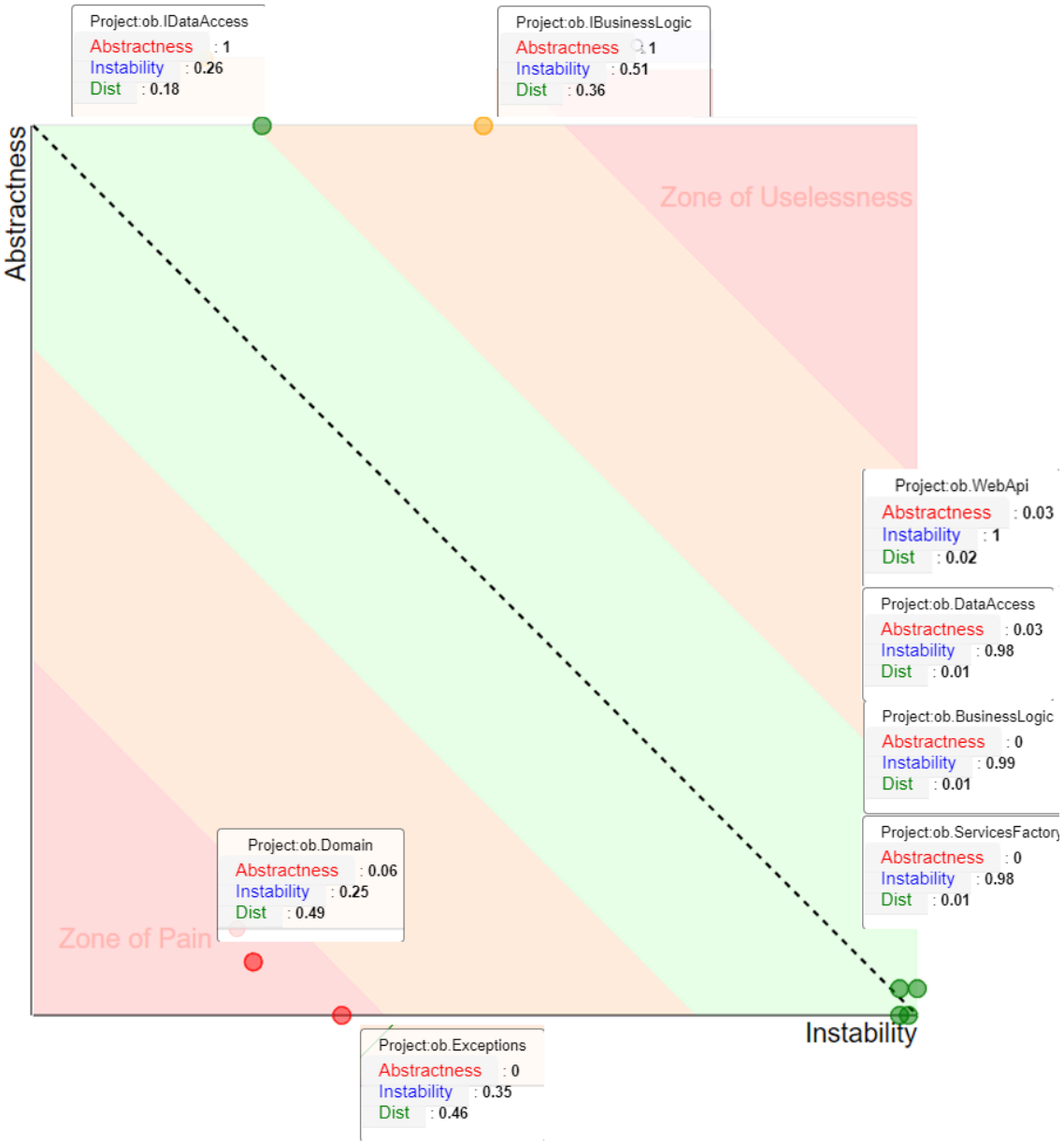
- **Dependencias entre Paquetes:** Los paquetes principales del sistema (ob.WebApi, ob.BusinessLogic, ob.DataAccess, y ob.Domain) están bien definidos y se relacionan de manera clara, evitando dependencias circulares que complicarían el mantenimiento del sistema.
- **Acoplamiento y Cohesión:** Los paquetes tienen niveles de acoplamiento y cohesión que reflejan una separación adecuada de responsabilidades. Por ejemplo, ob.WebApi interactúa con la capa de lógica de negocio (ob.BusinessLogic) y de acceso a datos (ob.DataAccess), lo cual es esperado en una arquitectura de este tipo.

Principios de Dependencias Estables:

- **Inestabilidad y Dependencias:** Los paquetes más inestables (ob.DataAccess y ob.BusinessLogic) dependen de paquetes más estables como ob.Domain. Esto sigue el principio de depender de componentes estables, asegurando que los componentes centrales del sistema no se vean afectados por cambios frecuentes.
- **Ausencia de Ciclos de Dependencia:** La falta de ciclos en el diagrama asegura que el sistema es más fácil de mantener y extender. Esto significa que podemos realizar cambios en un paquete sin riesgo de introducir efectos en cadena impredecibles en otros paquetes.

Mejoras Potenciales:

- **Revisión de Dependencias:** Aunque el sistema no tiene ciclos de dependencia, siempre es útil revisar las dependencias para asegurarse de que no haya dependencias innecesarias o redundantes.
- **Uso de Interfaces:** Aumentar el uso de interfaces puede ayudar a reducir la dependencia directa entre paquetes y facilitar el intercambio de implementaciones.



Análisis del Diagrama de Abstracción e Inestabilidad

El diagrama de abstracción e inestabilidad muestra cómo los diferentes componentes del sistema se distribuyen en función de su abstracción y estabilidad. Aquí se detallan las observaciones y conclusiones principales:

Puntos Clave:

1. **Componentes Inestables y Concretos:**
    - Los paquetes como ob.WebApi, ob.DataAccess, y ob.BusinessLogic son inestables y concretos (baja abstracción, alta inestabilidad). Esto significa que estos componentes cambian con frecuencia y no dependen de interfaces abstractas, lo cual es normal para componentes que manejan la lógica principal y las interacciones directas con los datos.

2. **ob.Domain:**
    - Este paquete es bastante estable (baja inestabilidad) porque muchos otros componentes dependen de él, pero está en la "zona de dolor" debido a su baja abstracción. Sin embargo, esto está bien porque ob.Domain contiene tipos y entidades fundamentales del negocio que no necesitan ser abstractos.
3. **Interfaces:**
    - Los paquetes ob.IDataAccess y ob.IBusinessLogic están en el punto medio con alta abstracción (1) y estabilidad moderada. Estos componentes actúan como contratos para otras partes del sistema, facilitando la extensibilidad y la implementación de diferentes estrategias sin modificar las interfaces base.

Conclusiones Basadas en Principios de Diseño:

- **Clausura Común y Reuso Común:** Los paquetes concretos e inestables deben agrupar funcionalidades que cambian por las mismas razones, lo que es adecuado para los servicios y controladores que manejan la lógica de negocio y la interacción con la base de datos.
- **Abstracciones Estables y Dependencias Estables:** Es clave que los componentes abstractos sean estables para que sirvan como fundamentos sólidos del sistema. ob.IDataAccess y ob.IBusinessLogic cumplen bien este principio, teniendo interfaces estables sobre las que se pueden construir implementaciones concretas.
- **Métrica de Distancia (D):** Los componentes deben idealmente estar cerca de la línea de equilibrio diagonal. Los componentes como ob.Domain están más alejados, indicando una necesidad de manejo cuidadoso para evitar la complejidad y la rigidez excesiva.

Resumen:

El análisis muestra que, aunque algunas áreas podrían beneficiarse de más abstracción para ser más flexibles, la estructura actual del sistema está bien para lo que necesitamos. Los componentes críticos del dominio son estables, y las interfaces nos dan una base sólida para extender el sistema. Las áreas en la "zona de dolor", como ob.Domain, están ahí porque contienen tipos fundamentales que no necesitan ser abstractos, lo cual está justificado por su rol en el sistema. En general, el diseño cumple con los principios básicos y funciona bien para su propósito.

Análisis del Diagrama de Métricas

Acoplamiento Aferente (Afferent Coupling):

- Indica cuántos otros paquetes dependen del paquete en cuestión.
- Un alto valor de acoplamiento aferente indica que muchos otros paquetes dependen de este, sugiriendo que es un componente central o crítico del sistema.

Acoplamiento Efferente (Efferent Coupling):

- Indica cuántas dependencias tiene el paquete hacia otros paquetes.
- Un alto valor de acoplamiento efferente indica que el paquete depende de muchos otros, lo cual puede sugerir una alta complejidad y una fuerte dependencia externa.

Cohesión Relacional (Relational Cohesion):

- Mide el grado en que las clases dentro del paquete están relacionadas entre sí.
- Una alta cohesión indica que las clases en el paquete están estrechamente relacionadas y trabajan juntas, lo cual es deseable para mantener un diseño limpio y modular.

Assemblies	# lines of code	# IL instruction	# Types	# Abstract Types	# lines of comment	% Comment	% Coverage	Afferent Coupling	Efferent Coupling	Relational Cohesion	Instability	Abstractness	Distance
ob.Domain v1.0.0.0	202	1008	16	1	0	0	-	54	18	1.5	0.25	0.06	0.49
ob.IDataAccess v1.0.0.0	0	0	2	2	-	-	-	23	8	1	0.26	1	0.18
ob.DataAccess v1.0.0.0	641	9942	29	1	48	6.97	-	2	127	1.03	0.98	0.03	0.01
ob.IBusinessLogic v1.0.0.0	0	0	12	12	-	-	-	25	26	0.08	0.51	1	0.36
ob.Exceptions v1.0.0.0	8	48	6	0	0	0	-	11	6	0.17	0.35	0	0.46
ob.BusinessLogic v1.0.0.0	662	5779	12	0	14	2.07	-	1	76	0.08	0.99	0	0.01
ob.ServicesFactory v1.0.0.0	27	123	1	0	0	0	-	1	60	1	0.98	0	0.01
ob.WebApi v1.0.0.0	446	3319	33	1	22	4.7	-	0	139	1.88	1	0.03	0.02

1. ob.Domain:

- **Afferent Coupling:** 54
- **Efferent Coupling:** 18
- **Relational Cohesion:** 1.5
- **Instability:** 0.25

Explicación: ob.Domain tiene un alto acoplamiento aferente, lo que significa que muchos otros paquetes dependen de él. Esto es normal ya que contiene los tipos y entidades principales del

negocio. Su acoplamiento efferente es moderado, lo que está bien porque no depende mucho de otros paquetes. La cohesión relacional es buena, lo que significa que las clases dentro de este paquete están bien relacionadas. Su baja inestabilidad es positiva porque los cambios en este paquete no afectan mucho a otros.

2. ob.DataAccess:

- **Afferent Coupling:** 2
- **Efferent Coupling:** 127
- **Relational Cohesion:** 1.03
- **Instability:** 0.98

Explicación: ob.DataAccess tiene un bajo acoplamiento afferente, indicando que pocos paquetes dependen de él, pero un muy alto acoplamiento efferente, lo que refleja que depende de muchos otros paquetes. Esto es típico para una capa de acceso a datos que interactúa con muchas entidades y servicios. La cohesión relacional es adecuada, y su alta inestabilidad sugiere que está expuesta a cambios frecuentes, lo cual es esperado debido a su naturaleza.

3. ob.IBusinessLogic:

- **Afferent Coupling:** 25
- **Efferent Coupling:** 26
- **Relational Cohesion:** 0.08
- **Instability:** 0.51

Explicación: ob.IBusinessLogic tiene un balance entre acoplamiento afferente y efferente, indicando que es una interfaz central en el sistema. La baja cohesión relacional es esperada en una interfaz que define contratos sin mucha lógica implementada. Su inestabilidad moderada es apta para una interfaz que debe ser flexible y extensible.

4. ob.BusinessLogic:

- **Afferent Coupling:** 1
- **Efferent Coupling:** 76
- **Relational Cohesion:** 0.08
- **Instability:** 0.99

Explicación: ob.BusinessLogic muestra un bajo acoplamiento afferente y un alto acoplamiento efferente, lo que muestra que depende de muchos otros paquetes. La baja cohesión relacional sugiere que podría beneficiarse de una mejor organización interna. La alta inestabilidad es una preocupación, indicando que este paquete está muy expuesto a cambios, lo que puede ser problemático.

5. ob.WebApi:

- **Afferent Coupling:** 0
- **Efferent Coupling:** 139

- **Relational Cohesion:** 1.88
- **Instability:** 1

Explicación: ob.WebApi no tiene acoplamiento afferente, lo cual es esperado ya que es la capa de presentación. Su alto acoplamiento efferente dice que depende de muchas otras partes del sistema. La alta cohesión relacional es positiva, ya que muestra que las clases dentro de este paquete están bien relacionadas. La inestabilidad máxima es típica para una capa que debe adaptarse rápido a cambios en las necesidades del usuario.

Principios de Dependencias Estables

- **Dependencias Estables:** Los componentes como ob.Domain y ob.IBusinessLogic son estables y proporcionan una base sólida sobre la cual otros componentes pueden depender. Esto es crucial para mantener un sistema robusto y extensible.
- **Inestabilidad y Dependencias:** Paquetes como ob.BusinessLogic y ob.DataAccess muestran alta inestabilidad, lo que significa que dependen de muchos otros componentes. Aunque esto puede ser esperado en algunas capas, es importante manejar con cuidado estas dependencias para evitar una arquitectura frágil.

Conclusión

Aunque algunos paquetes presentan alta inestabilidad y acoplamiento, esto está justificado por su rol en el sistema. No quitando, siempre hay margen para mejorar, como introducir más abstracciones para reducir dependencias y mejorar la cohesión interna. Esto asegura que nuestro diseño se mantenga robusto y flexible, así yendo de la mano con los principios de dependencias estables y buenas prácticas de diseño de software.

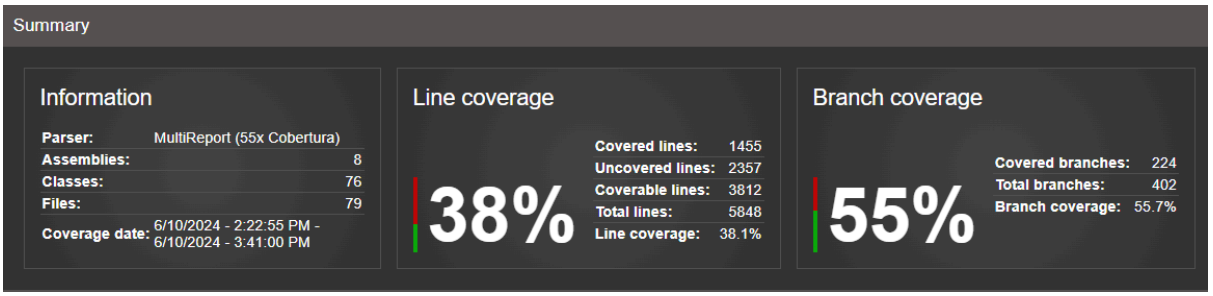
Resumen de las Mejoras al Diseño

1. **Optimización de la Interfaz de Usuario (UI)**:
    - Mejoramos la UI utilizando Angular para crear una aplicación de página única (SPA). Esto hace que la experiencia de usuario sea más fluida e intuitiva. La SPA carga contenido dinámicamente sin recargar la página completa, mejorando la velocidad y eficiencia de navegación. Además, implementamos un diseño responsivo que se adapta a diferentes tamaños de pantalla y dispositivos, lo que mejora la accesibilidad y la experiencia general del usuario.
2. **Gestión de Roles, Permisos y Extensibilidad del Sistema**:
    - Mejoramos la gestión de roles permitiendo la invitación de futuros administradores de empresas constructoras, además de encargados. También añadimos un mecanismo para importar edificios desde sistemas externos mediante adaptadores desarrollados por terceros. Esto permite la integración con múltiples formatos de datos (JSON, XML, CSV), asegurando que el sistema pueda adaptarse y crecer según las necesidades futuras sin grandes cambios en el código base.
3. **Manejo de Múltiples Edificios por Personal de Mantenimiento y Reportes Detallados**:
    - Ahora, los usuarios de mantenimiento pueden gestionar múltiples edificios, mejorando la eficiencia operativa. Además, añadimos la funcionalidad para que los encargados generen reportes detallados de solicitudes por apartamento, proporcionando una visión clara y organizada de las necesidades de mantenimiento de cada propiedad. Esto permite que el sistema soporte un mayor número de operaciones simultáneas y distribuidas, adaptándose mejor a las necesidades de organizaciones más grandes.
4. **Importación Dinámica de Edificios y Teoría de Extensibilidad**:
    - Introdujimos la capacidad de importar datos de edificios desde sistemas externos utilizando adaptadores de terceros. Esto facilita la integración de múltiples formatos como XML, JSON y CSV, y permite la expansión continua del sistema sin necesidad de recompilación. Este enfoque modular y extensible se alinea con los principios de diseño de software desacoplado, promoviendo la extensibilidad y la mantenibilidad a largo plazo. Al permitir que terceros desarrollen adaptadores de importación, garantizamos que el sistema puede evolucionar y adaptarse a nuevos requisitos y tecnologías sin alterar la base del código existente.

Estas mejoras no solo optimizan la usabilidad y flexibilidad del sistema, sino que también aseguran su mantenibilidad a largo plazo. La aplicación de principios de diseño y patrones nos permitió crear una arquitectura robusta y escalable, capaz de satisfacer las demandas crecientes y diversificadas del mercado internacional.

Anexo

Informe de Cobertura de Tests



En esta sección, analizamos la cobertura lograda por nuestras pruebas, discutiendo en forma crítica su efectividad. La cobertura de pruebas es una métrica crucial que nos permite evaluar el alcance y la exhaustividad de nuestras pruebas unitarias y de integración. Una alta cobertura indica que una gran parte del código ha sido evaluada mediante pruebas, lo que ayuda a detectar y corregir errores, y asegura que los cambios futuros no introduzcan nuevos problemas.

Aunque el porcentaje de Line Coverage a primera vista parezca bastante bajo, vamos a desglosar todo para entender bien qué es lo que nos quiere decir.

El análisis de cobertura se ha realizado utilizando una herramienta específica que mide qué partes del código han sido ejecutadas durante la ejecución de las pruebas. A continuación, presentamos los resultados obtenidos, organizados por los principales componentes del sistema.

ob.BusinessLogic: 67.7%

ob.BusinessLogic	684	326	1010	1511	67.7%
ob.BusinessLogic.AdminConstructoraService	188	66	254	334	74%
ob.BusinessLogic.AdminService	31	9	40	73	77.5%
ob.BusinessLogic.CategoriaService	11	13	24	46	45.8%
ob.BusinessLogic.ConstructoraService	34	0	34	57	100%
ob.BusinessLogic.DeptoService	42	20	62	93	67.7%
ob.BusinessLogic.DuenoService	22	2	24	43	91.6%
ob.BusinessLogic.EdificioService	74	23	97	147	76.2%
ob.BusinessLogic.EncargadoService	125	105	230	344	54.3%
ob.BusinessLogic.InvitacionService	45	5	50	76	90%
ob.BusinessLogic.MantenimientoService	51	15	66	101	77.2%
ob.BusinessLogic.SessionService	38	6	44	75	86.3%
ob.BusinessLogic.SolicitudService	23	62	85	122	27%

La cobertura de pruebas en la capa de lógica de negocio es relativamente alta, lo que sugiere que la mayoría de las funcionalidades críticas han sido evaluadas. No obstante, hay áreas que no están completamente cubiertas, lo que podría ser una oportunidad para mejorar nuestras pruebas.

ob.DataAccess: 8%

ob.DataAccess	146	1660	1806	2231	98.7%
ob.DataAccess.AppContext	87	13	100	150	87%
ob.DataAccess.CategoriaRepository	4	0	4	16	100%
ob.DataAccess.ConstructoraRepository	4	0	4	17	100%
ob.DataAccess.DeptoRepository	4	0	4	13	100%
ob.DataAccess.DuenoRepository	4	0	4	13	100%
ob.DataAccess.EdificioRepository	4	0	4	13	100%
ob.DataAccess.InvitacionRepository	4	0	4	13	100%
ob.DataAccess.SessionRepository	11	0	11	21	100%
ob.DataAccess.SolicitudRepository	4	0	4	13	100%
ob.DataAccess.UsuarioRepository	7	0	7	17	100%

La cobertura de pruebas en la capa ob.DataAccess es alta, con un 98.7% de líneas cubiertas. AppContext tiene una cobertura del 87%, indicando que la mayoría de las configuraciones están probadas. Los repositorios como CategoriaRepository, ConstructoraRepository, y otros tienen una cobertura del 100%, asegurando que las operaciones CRUD están completamente cubiertas. Es importante seguir mejorando la cobertura en AppContext para alcanzar una cobertura total y mantener la alta cobertura en los repositorios para asegurar la fiabilidad del sistema de acceso a datos.

ob.Domain: 95.6%

ob.Domain	264	12	276	550	95.6%
ob.Domain.AdminConstructora	4	0	4	14	100%
ob.Domain.Administrador	3	0	3	13	100%
ob.Domain.Categoria	6	0	6	19	100%
ob.Domain.Constructora	12	0	12	25	100%
ob.Domain.Depto	25	0	25	59	100%
ob.Domain.Dueno	24	0	24	48	100%
ob.Domain.Edificio	43	0	43	81	100%
ob.Domain.Encargado	5	0	5	12	100%
ob.Domain.Invitacion	25	0	25	49	100%
ob.Domain.Mantenimiento	3	0	3	10	100%
ob.Domain.Session	6	1	7	13	85.7%
ob.Domain.Solicitud	49	4	53	91	92.4%
ob.Domain.Usuario	26	0	26	47	100%
ob.Domain.Validator	33	7	40	69	82.5%

La cobertura de pruebas en el paquete ob.Domain es bastante alta, alcanzando un 95.6%. La mayoría de las clases, como AdminConstructora, Administrador, Categoria, Constructora, y otras, tienen una cobertura del 100%, lo que significa que probamos casi todas sus funcionalidades. Sin embargo, hay algunas clases con menor cobertura, como Session (85.7%) y Validator (82.5%). Aunque la mayoría

de nuestro código está bien cubierto, todavia podemos mejorar las pruebas en estas áreas para asegurar que todo funcione correctamente.

ob.Exceptions y ob.ServicesFactory

+ ob.Exceptions	6	18	24	79	25%
+ ob.ServicesFactory	0	37	37	67	0%

La cobertura de pruebas en los paquetes ob.ServicesFactory y ob.Exceptions es baja porque estos componentes generalmente manejan aspectos de configuración y manejo de errores que no siempre requieren pruebas exhaustivas. ob.ServicesFactory se encarga de inicializar y configurar servicios, donde la lógica es simple y a menudo estática. En cuanto a ob.Exceptions, este paquete maneja excepciones personalizadas, y la lógica principal de estas clases suele ser trivial, confiando en pruebas más específicas a nivel de integración donde se manejan estas excepciones en contextos de uso real.

ob.WebApi: 53.8%

− ob.WebApi	355	304	659	1410	53.8%
ActionFilter	0	6	6	14	0%
Program	0	41	41	61	0%
ResourceFilter	0	6	6	14	0%
ResultFilter	0	6	6	14	0%
ob.WebApi.Controllers.AdminConstructoraController	53	42	95	215	55.7%
ob.WebApi.Controllers.AdministradorController	25	5	30	84	83.3%
ob.WebApi.Controllers.BaseController	12	0	12	29	100%
ob.WebApi.Controllers.CategoriaController	11	0	11	41	100%
ob.WebApi.Controllers.ConstructoraController	11	0	11	43	100%
ob.WebApi.Controllers.DuenoController	12	0	12	43	100%
ob.WebApi.Controllers.EncargadoController	45	37	82	176	54.8%
ob.WebApi.Controllers.InvitacionController	8	0	8	34	100%
ob.WebApi.Controllers.MantenimientoController	12	10	22	64	54.5%
ob.WebApi.Controllers.SessionController	14	0	14	44	100%
ob.WebApi.DTOs.CategoriaDTO	5	1	6	15	83.3%
ob.WebApi.DTOs.ConstructoraDTO	12	1	13	23	92.3%
ob.WebApi.DTOs.DeptoDTO	30	5	35	48	85.7%
ob.WebApi.DTOs.DuenoDTO	10	0	10	20	100%
ob.WebApi.DTOs.EdificioCreateDTO	8	0	8	20	100%
ob.WebApi.DTOs.EdificioDTO	25	12	37	58	67.5%
ob.WebApi.DTOs.EncargadoDTO	13	3	16	30	81.2%
ob.WebApi.DTOs.InvitacionDTO	5	7	12	24	41.6%
ob.WebApi.DTOs.SolicitudDTO	38	12	50	83	76%
ob.WebApi.DTOs.UserLoginModel	2	0	2	8	100%
ob.WebApi.DTOs.UsuarioCreateModel	4	12	16	34	25%
ob.WebApi.Filters.AuthenticationFilter	0	25	25	47	0%
ob.WebApi.Filters.AuthorizationFilter	0	42	42	69	0%
ob.WebApi.Filters.ExceptionFilter	0	31	31	55	0%

La cobertura de pruebas en el paquete ob.WebApi es bastante completa, aunque maneje una cobertura global del 53.8%. La mayoría de los controladores, como AdminController, CategoriaController, y ConstructoraController, tienen una cobertura alta, asegurando que las principales rutas y funcionalidades están probadas. Sin embargo, algunas funciones y filtros que no requieren pruebas exhaustivas, como ActionFilter y ResourceFilter, tienen baja o nula cobertura, lo que disminuye el porcentaje general. Estas áreas no probadas son típicamente componentes de configuración o manejo de errores, que no afectan significativamente la robustez del sistema. También tiene algunos aspectos a mejorar en archivos como EncargadoController e InvitacionDTO

Tabla de Cambios - API

API	DESCRIPCIÓN	REQUEST BODY	RESPONSE BODY	EXPLICACIÓN
POST /api/administradores	Crear un nuevo administrador	{ "nombre": "Juan", "apellido": "Perez", "email": "juan.perez@example.com", "contraseña": "segura123", "rol": "admin" }	{ "status": 201, "data": { "id": 1, "email": "juan.perez@example.com", "rol": "admin" } }	Se añadió el campo rol para especificar si el administrador es un admin general o un admin de empresa constructora.
POST /api/invitaciones	Invitar a un nuevo usuario a la plataforma	{ "email": "ejemplo@gmail.com", "nombre": "Ana", "fechaLimite": "2024-05-30", "rol": "admin" }	{ "status": 201, "data": { "id": 1, "email": "ejemplo@gmail.com" } }	Se añadió el campo rol para especificar si el invitado será un encargado o un admin de empresa constructora.
POST /api/edificios	Crear un nuevo edificio	{ "nombre": "Edificio Central", "direccion": "123 Calle Principal", "ubicacion": "Centro Ciudad", "empresaConstructora": "Construcciones ABC", "gastosComunes": 1200, "departamentos": [{ "piso": 1, "numero": 101, "dueño": {"nombre": "Ana", "apellido": "Ruiz", "email": "ana.ruiz@example.com"}, "cantidadCuartos":	{ "status": 201, "data": { "id": 1, "nombre": "Edificio Central" } }	Ahora solo los admins de empresa constructora pueden crear edificios.

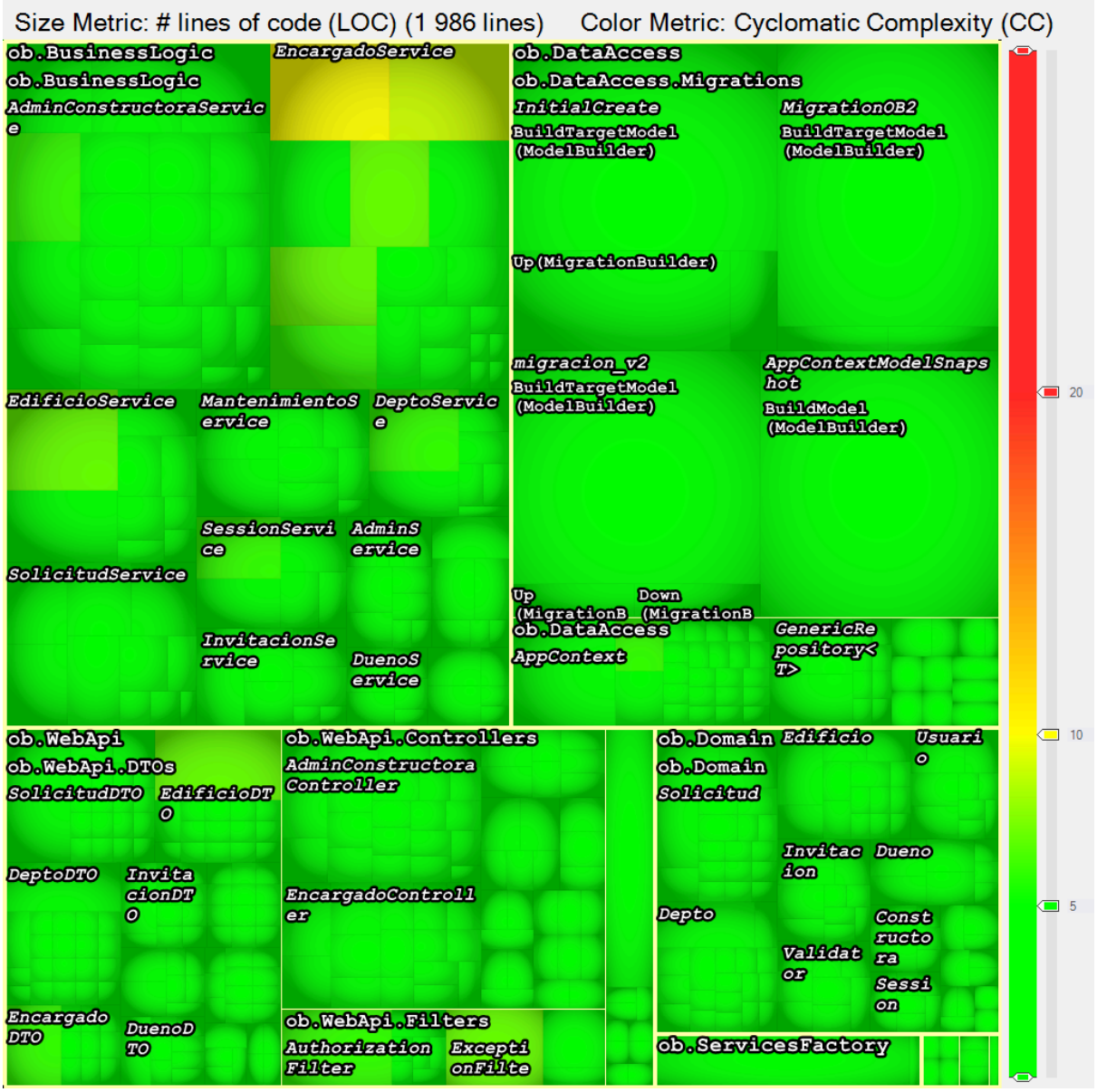
		3, "cantidadBaños": 2, "terraza": true }] }		
PUT /api/edificios/{id}	Actualizar detalles de un edificio	{ "nombre": "Edificio Central Mod", "direccion": "123 Calle Principal", "ubicacion": "Centro Ciudad", "empresaConstructora": "Construcciones ABC Mod", "gastosComunes": 1300, "departamentos": [{ "piso": 1, "numero": 101, "dueño": {"nombre": "Ana", "apellido": "Ruiz", "email": "ana.ruiz@example.com"}, "cantidadCuartos": 3, "cantidadBaños": 2, "terraza": true }] }	{ "status": 200, "data": { "id": 1, "nombre": "Edificio Central Mod" } }	Se añadió la capacidad de actualizar la lista de departamentos asociados al edificio.
POST /api/usuarios/mantenimiento	Crear un usuario de mantenimiento	{ "nombre": "Carlos", "apellido": "Constructor", "email": "carlos.constructor@example.com", "contraseña": "construir123", "edificios": [1, 2, 3] }	{ "status": 201, "data": { "id": 1, "email": "carlos.constructor@example.com" } }	Ahora los usuarios de mantenimiento pueden atender múltiples edificios.
GET /api/solicitudes	Obtener todas las solicitudes, filtrable por categoría	{ "categoriaId": 2 }	{ "status": 200, "data": [{"id": 1, "descripcion": "Fuga de agua", "estado":	Se añadió la capacidad de filtrar solicitudes por categoría, incluyendo subcategorías.

			"abierto", "categoria": "Fontanero"}] }	
POST /api/importar	Importar edificios desde un archivo	{ "tipo": "json", "archivo": "path/to/file.json" }	{ "status": 200, "data": "Importación exitosa" }	Nueva API para importar edificios desde archivos JSON o XML utilizando importadores desarrollados por terceros.

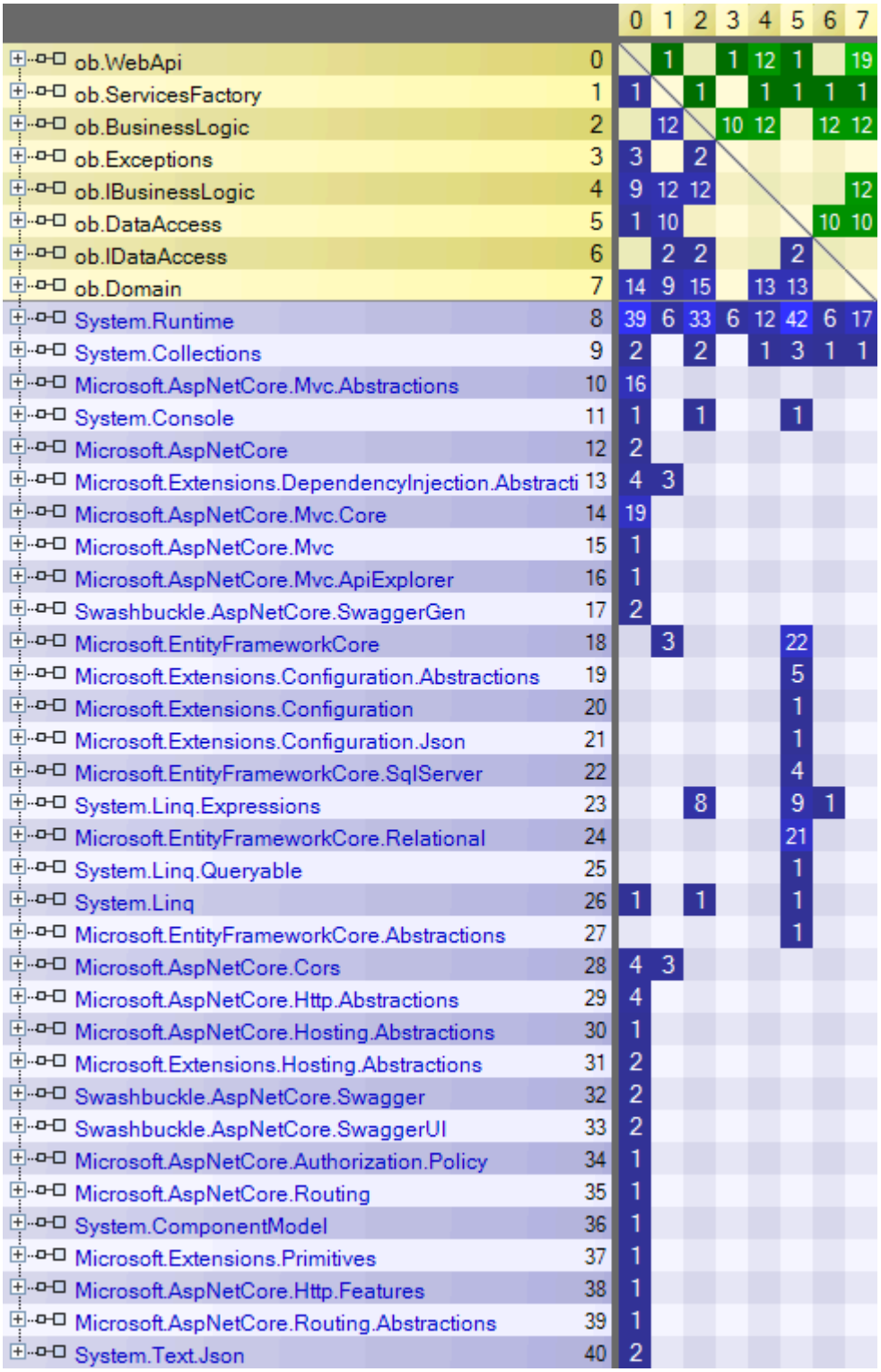
Explicaciones de los Cambios

1. **Añadir Campos Nuevos**:
   - **POST /api/administradores y POST /api/invitaciones**: Se añadió el campo rol para especificar si el usuario invitado o creado será un encargado o un admin de empresa constructora. Esto permite una mejor gestión de roles y permisos en la plataforma.
2. **Modificar Estructuras Existentes**:
   - **POST /api/edificios y PUT /api/edificios/{id}**: Ahora, solo los admins de empresa constructora pueden crear edificios y se añadió la capacidad de actualizar la lista de departamentos asociados al edificio. Estos cambios aseguran que la administración de edificios esté centralizada y sea más controlada.
3. **Nuevas Funcionalidades**:
   - **POST /api/importar**: Se implementó una nueva API para permitir la importación de edificios desde archivos JSON o XML. Esta funcionalidad es esencial para la internacionalización y facilita la integración con sistemas externos, permitiendo que terceros desarrollen importadores personalizados.
4. **Manejo de Múltiples Edificios**:
   - **POST /api/usuarios/mantenimiento**: Ahora, los usuarios de mantenimiento pueden atender múltiples edificios, lo cual mejora la flexibilidad y eficiencia en la gestión de servicios de mantenimiento.

Mapa de Árbol Metrico



Matriz de Dependencia



		0	1	2	3	4	5	6	7
ob.WebApi	0		1		1	12	1		19
ob.ServicesFactory	1	1		1		1	1	1	1
ob.BusinessLogic	2		12		10	12		12	12
ob.Exceptions	3	3		2					
ob.IBusinessLogic	4	9	12	12					12
ob.DataAccess	5	1	10					10	10
ob.IDataAccess	6		2	2			2		
ob.Domain	7	14	9	15		13	13		
System.Runtime	8	39	6	33	6	12	42	6	17
System.Collections	9	2		2		1	3	1	1
Microsoft.AspNetCore.Mvc.Abstractions	10	16							
System.Console	11	1		1			1		
Microsoft.AspNetCore	12	2							
Microsoft.Extensions.DependencyInjection.Abstracti	13	4	3						
Microsoft.AspNetCore.Mvc.Core	14	19							
Microsoft.AspNetCore.Mvc	15	1							
Microsoft.AspNetCore.Mvc.ApiExplorer	16	1							
Swashbuckle.AspNetCore.SwaggerGen	17	2							
Microsoft.EntityFrameworkCore	18		3				22		
Microsoft.Extensions.Configuration.Abstractions	19						5		
Microsoft.Extensions.Configuration	20						1		
Microsoft.Extensions.Configuration.Json	21						1		
Microsoft.EntityFrameworkCore.SqlServer	22						4		
System.Linq.Expressions	23			8			9	1	
Microsoft.EntityFrameworkCore.Relational	24						21		
System.Linq.Queryable	25						1		
System.Linq	26	1		1			1		
Microsoft.EntityFrameworkCore.Abstractions	27						1		
Microsoft.AspNetCore.Cors	28	4	3						
Microsoft.AspNetCore.Http.Abstractions	29	4							
Microsoft.AspNetCore.Hosting.Abstractions	30	1							
Microsoft.Extensions.Hosting.Abstractions	31	2							
Swashbuckle.AspNetCore.Swagger	32	2							
Swashbuckle.AspNetCore.SwaggerUI	33	2							
Microsoft.AspNetCore.Authorization.Policy	34	1							
Microsoft.AspNetCore.Routing	35	1							
System.ComponentModel	36	1							
Microsoft.Extensions.Primitives	37	1							
Microsoft.AspNetCore.Http.Features	38	1							
Microsoft.AspNetCore.Routing.Abstractions	39	1							
System.Text.Json	40	2							

Métricas de Aplicación